Tampere University

Fouzia Khan

Microservices Metrics Visualization

Faculty of Information Technology and Communication Sciences (ITC)
Master's thesis
November 2020

Abstract

Fouzia Khan: Microservices Metrics Visualization
Master's thesis
Tampere University
Master's Degree Programme in Software Development
November 2020

Many companies are using a distributed approach where systems can be developed in smaller chunks of functionalities called microservices. Due to their smaller size, microservices offer benefits such as smaller development teams, independent choice of developmental technologies and a lesser time to market to name a few. However, as the systems grow bigger, the number of microservices can increase up to hundreds and thousands which makes all this more challenging for the development teams and the project managers to analyse the services performance, prioritize resources and the ability to see the overall picture from business and management perspective. This thesis aims to look for a possibility to develop a tool by combining existing open source tools which could help overcome this challenge. The thesis aims to research and explore various existing open source microservices monitoring and tracing tools to figure out if they could be utilized to develop a microservices visualization tool comprised of a dynamic call graph displaying performance and business metrics for individual microservices. These tools are individually explored and run with test projects to verify their potential and the output is carefully analysed before deciding if the tool should be included in the implementation. The results show that such a tool can be developed by using some existing open source tools such as Jaeger, Prometheus and Grafana. The implemented tool is capable of constructing a microservices dynamic dependency network graph which includes various metrics e.g. number of calls being made from one microservice to the other, average response time per service, average load per minute per service, ratio of open and closed issues, ratio of open and closed bugs, ratio of cost and revenue and the amount of effort spent for each microservice in the system. This tool can make it easier for the developers and managers to visualize the number of calls being made and performance related challenges in microservices architecture-based systems only by looking at the graph and it can also help the business managers to make some strategic decisions on the basis of an overall picture of the system.

Keywords: Microservices, dynamic call graph, latency time, Jaeger, Prometheus, Grafana.

The originality of this thesis has been checked using the Turnitin Originality Check service.

PREFACE

I would like to thank my thesis Supervisor Davide Taibi, whose guide and support has helped me produce this work. And thanks to my sister for providing me the moral support.
The Master thesis implementation and writing has been a wonderful process for me. I have learned quite a lot about various tools and techniques in the subject matter which have a significant contribution to the skill-set I possess now.
Thanks to the examiners Davide Taibi and Francesco Lomio for their support.

Tampere, 12 November 2020
Fouzia Khan

Contents

1 Introduction

Over the period, technology has evolved at a very fast pace. Many improvements in methods to code and software architectures have been brought up in the Software development industry to improve the quality of developed products. For example, continuous delivery assists the software companies to deliver products faster to their customers. Similarly, automating the recurrent development and operational processes have reduced the overhead for developers. However, as the products grow and get complicated the size of the codebase increases and it becomes very difficult for the developers to debug issues or identify areas to modify code for additional features. Despite following the modular approach to keep various functionalities separate it turns very difficult to maintain the code quality and developer's productivity with a traditional monolithic approach. Therefore, the need to use microservices based systems arises.

Microservices based systems comprise several smaller and autonomous services communicating with each other. These services are called microservices. The microservices architectural system is divided into smaller functionalities and each microservice is developed to cater to one functionality. These services are independent and different development teams can work on them independently using the desired tools and technologies, hence, the code base remains smaller. Owing to the benefits of microservices based architecture, many big companies like Uber, Netflix, Amazon, eBay have adopted microservices based architecture and many other companies are planning to move towards it.

Despite having many advantages, one of the major challenges of using microservices is their complex communication and their dependencies upon each other. Currently, there are systems with hundreds and thousands of microservices communicating with each other and it has become very difficult for companies to make strategic and business decisions for such large and complex systems. It can often cause problems for the developers as well to find out problematic connections between the services and the performance issues. Hence, a microservices visualization tool is needed which could draw an overall services dependency graph and display some important metrics to make it easier to conclude business decisions e.g. on which service to invest time and money based on its usage, priority and identify services performance bottlenecks. Currently, there are some open source and commercial distributed tracing tools able to perform this task. However, the open source tools draw a very basic dependency graph of microservices with no additional metrics e.g. services latency time, load per minute, cost vs revenue and open vs closed issues and bugs.

Aim of this research is also to find a possibility to construct the microservices dynamic dependency call graph without manually instrumenting the code. However, as the instrumentation is inevitable, the research solely focuses on improving the output dependency graph from an existing tracing tool including number of calls made from one microservice to another, services latency time, load per minute, cost vs revenue, open vs closed issues, open vs closed bugs and total effort spent. The dependency graph should display different icons to represent databases, message buses and microservices. Finally, thesis researches the possibility to integrate the final call dependency graph into some existing metrics visualization tool commonly used by microservices architecture-based companies.

In chapter 2, the thesis covers a detailed literature review of related concepts, existing tools and methods used in the microservices monitoring and the tools capabilities. Chapter 3 draws a comparison between some popular monitoring and opentracing tools. Chapter 4 describes the MsViz tool which is the output of this thesis work wheras chapter 5 focuses on the implentation of MsViz tool and the tools and techniques it uses to provide the resultant output. Chapter 6 discusses the uncertainity factors and limitations of the tool. Chapter 7 comprises of thesis conclusion.

2 Background

This chapter covers the concepts related to microservices, their monitoring and how microservices communicate with each other. The chapter also includes the concepts of logging, monitoring, tracing, how they are different from each other and some opensource logging, monitoring and tracing tools. Each section is divided into subsections to describe that particular aspect in detail.

2.1 Microservices

Many server-side programming languages provide the functionality of modularization to keep the code abstraction level and separate the related chunks of functionality, however, the chunks of program remain highly dependable upon each other and cannot be managed separately and they have to share all the resources on the same machine. These kinds of programs are called monoliths. Monoliths pose various challenges and limitations such as difficulty to maintain larger code bases and updating libraries can create inconsistencies within the program. They can only be deployed as one single program regardless of different resource requirements of different groups of functionalities. The whole server hosting the application needs to be scaled up or down in case of increasing/decreasing requests, all the modules need to be developed using one server side or client-side language despite varying technological requirements for different modules. Moreover, as the number of features increase, the code bases grows larger and complex which makes it very challenging for the developers to add features and fix bugs. Different approaches such as cohesion and modularization have been adopted to avoid this problem. Despite using these approaches, the code-base cannot be well kept unless the same functionality of a system is grouped together. To deal with all these issues microservices based systems have been proposed, where one microservice handles only a small independent functionality and communicates with other microservices via messages or Rest APIs. Hence, the related knowledge remains between the same business boundary and the code-bases remain reasonable without growing too much. [9][19]

Microservices can exist separately and talk to other services in the system through network calls. They should be managed and deployed independently. If dependency between two microservices increase those microservices become decoupled and often pose challenges in deployments.

2.1.1 Characteristics of Microservices

Microservices based systems offer the following characterics over monlithic architetures [19]:

Heterogeneous: For a system of microservices, it is possible to choose the most suitable programming language for one particular microservice. This aspect is very useful in a case when some functionality sets of the system needs to excel in different attributes to perform well as compared to the other functionality. For example, a microservice which need better performance can be developed in a language which helps achieve this attribute. Besides this, heterogeneity can help try different or new programming languages and frameworks. In case of monolithic systems, it is quite hard to try a new technology as it imposes a huge risk onto the whole system. In case of microservices, new technologies can be used for the least important microservice to evaluate it's benefits before choosing for a larger chunk.

Resilient: As monolithic systems are not resilient, a failure in one component can break down the whole system. To solve this problem one can run the whole system in more than one machines so if the failure occurs in one, the other service keeps running. Whereas in microservices, resiliency avoids the breakage of the whole system by degrading systems functionality. However, to ensure the resilience, it is important to understand the possible failures first which can break the system.

Scaling: In monolithic systems if a smaller part of the systems needs scaling up, the whole system is to be scaled which can cause extra cost. As microservices comprise of smaller independent code of chunks, only the relevant services can be scaled up or down as per demand.

Easy Deployments: A major issue with monolithic systems is the frequent re-deployments and more down times. Once a change (smaller or larger) is made it requires the whole system to be re-deployed. Sometimes, the re-deployment package might contain plenty of new work or an entire new release which has a greater potential to induce risk to the working system. Once this new release is redeployed, it can be quite challenging to identify the problems, the areas where they occur and the re-deployment can be quite time consuming as well. However, the microservices based systems ease this deployment process and allow only deploying code to the microservice where it belongs to while keeping the other services unaffected. This practice also makes it easier to debug the issues in case they occur or even rollback the whole deployment in case required.

Organizational Alignment: The larger teams working on same code-bases can create problems, whereas the smaller teams working on a small code-bases can be more productive. Hence, microservices ease this kind of organizational structure where one team can be assigned only one microservice to improve the code under-

standing and the team productivity.

Composability: Composability in microservices allows the reuse of similar functionality. Monoliths used to focus on only one product channel, for example, whether it should work for the desktops or the mobiles but with the growing customer needs the products now need to meet the web, mobile phones, tablets and wearable devices with the use of single functionality. In case of a microservices based architecture it is possible to reuse an existing functionality for multiple channels.

Optimizing for Replace-ability: Changing or upgrading an old monolithic system to a newer technology is quite challenging and risky and due to this very reason some companies would keep their bigger systems in the old technologies for a long time as upgrading can be very costly and risky in itself. Whereas, it is quite easy to experiment with the newer technologies in case of microservices. Due to their small size and independence, the experiment can pose least threat to the other microservices in the system.

2.1.2 Microservices Architecture

Various benefits can be achieved by utilizing the microservices based architecture over monolithic architecture because microservices are resilient, heterogeneous, scalable, independently deployable and can help developers increase their productivity as the code-base remains smaller. [1] Today many companies have chosen to switch to microservices based architecture, for example, Netflix is using 500+ microservices for its online service system which is handling approximately 2 billion API requests each day. Tencent'sWeChat owns 2000 microservices. These services often communicate asynchronously. Having such a complex network of microservices makes it difficult to handle these systems as just a single page load might be invoking multiple microservices. Microservices interact with each other more frequently and scalability might result in having thousands of physical instances being managed by some service discovery tool e.g. docker swarm. Hence, it is important to address issues like communication, failure cascade, discovery and authentication.[2]

To create a system of microservices, some of the key components to consider from an architectural perspective are services containerization, modes of communication and a service discovery mechanism. A Linux or Docker container is a lightweight virtual machine which provides all the necessary resources e.g. CPU, memory, block IO and network resources for a microservice to run. Another important architectural component is server discovery. As the network locations of individual microservices change, it is required that their locations are known by other microservices. To deal with this a service discovery mechanism is used. Server discovery uses a component called service registry which keeps a database of all the registered services. Service registry uses two APIs called management API and query API. Management API is

used by the services to register or de-register themselves and query API is used to make requests to a known service. Client side and server-side discovery are the two patterns of service discovery. In client-side discovery services use service registry to query an instance whereas in server-side discovery services use a router to send a service request to the service registry which forwards the request to the actual service instance. In the end, microservices based architectures cannot overcome the challenges if the requirements are incorrect as it will lead to the overall architectural decay. On the other hand, excellent DevOps skills are needed to monitor and deploy such systems. [10]

Another challenge to deal with microservices is their partitioned database architecture which is needed for transactions that need to update multiple databases which could belong to various microservices in a system. Testing microservices systems is also very challenging as compared to monoliths. For testing a whole business flow the entire system needs to be in place whereas an alternative is to create the stubs for unavailable services for the test use. [11]

Figure 2.1 shows how service registry acts in microservices based architecture (MSBA).

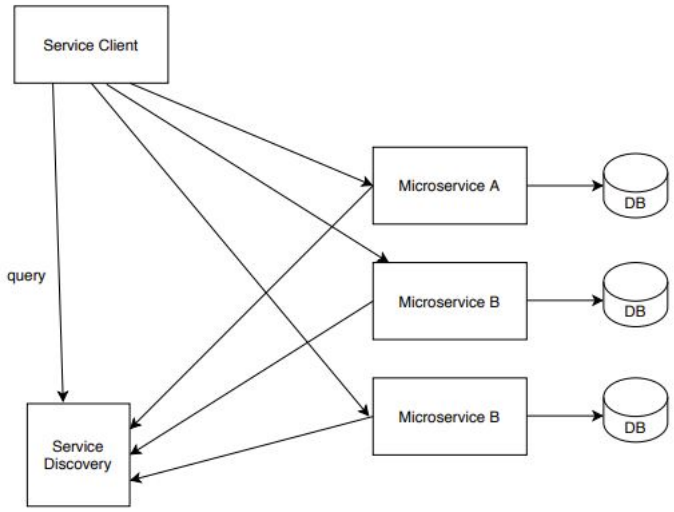

Figure 2.1 *Service Discovery in MSBA*

2.1.3 Communication Modes

In microservices architecture one important component to consider is how the services should communicate. Communication can be synchronous or asynchronous. In synchronous communication a service sends a request and waits for the response before starting any other operation. In asynchronous communication, the microservice sends a request but does not block further operations before a response is

arrived. Publish/subscribe is another type of communication where a microservice publishes a message and waits for the reply for a certain amount of time, meanwhile, another microservice in the same network can read or consume this message. The synchronous communication mechanism is based on HTTP Rest APIs or Thrift whereas asynchronous communication mechanism can be Advanced Message Queuing Protocol (AMQP). Microservices usually use a combination of these interaction mechanisms. There are some asynchronous message-based open source systems which developers can use to implement microservices communication mechanism such as RabbitMQ, Apache Kafka and Apache ActiveMQ. [10]

2.2 Microservices Inter Communication

As microservices are developed in a programming language independent from each other, they also need a programming language independent call mechanism for communication with other microservices. This call mechanism can be synchronous or synchronous. The synchronous mechanism works using HTTP API whereas asynchronous messaging protocol includes systems such as RabbitMQ, ActiveMQ and Kafka. [10]

Messaging protocol such as message buses provide a way of communication between independent microservices and are called message brokers. As the microservices are developed and deploy independently, they need to talk to each other through messages which can be requests, replies or chunks of information. Message bus provides a platform for these messages to stay in a queue for a while before they are addressed and responded. Message queues provide a one way system in which one application sends a request to another and receives the response if it is expected. The message sending client or service is called publisher whereas the message receiving side is called a consumer. Message queuing is synchronous way of communication, hence, it does not wait for the response and continues running other threads and processes it is supposed to execute. One of the most important function of message broker is its ability to fault tolerate and scale up in case of increase in load. [13]

Just like microservices are loosely coupled, the broker message communication system corresponds to the same pattern. Microservices do not need to know each others location also whether other services are up or not. Message broker knows how to communicate with other microservices and keeps their location. A microservice can merely send the message to another regardless of its status and location. If the receiving service is down, the messages can stay in the queue for a while until the service is up again and ready to consume the messages. Moreover, the producer, consumer instances of a message broker can be scaled up or down depending on the load. So far,various message based communication solutions have been proposed.

Some propriety tools are IBM, WebSphere MQ and Microsoft message queuing. Others are opensource tools such as RabbitMQ, ActiveMQ, JBoss. [13]

In the subsequent subsection three common asynchronous communication technologies are discussed.

2.2.1 Kafka As Message Bus

Kafka is mainly used for streaming data in enterprise infrastructures. External systems can be connected to Kafka to transmit messages or data through Kafka connect. Kafka is also able to process streams of data between producers and consumers. A producer can produce messages or Kafka streams and publish those to the Kafka topics from where messages can be consumed by Kafka consumers. Each topic can be divided into partitions. Kafka brokers are responsible for storing incoming messages in partitions and allow consumers to read and consume these messages. As Kafka is capable of handling streams of data it can be deployed on multiple servers as a cluster. To start a Kafka server, it is required to start Zookeeper first which acts as a service discovery tool and is responsible to manage Kafka. Zookeeper keeps track of active producers, consumers, brokers, topics and partitions. In case of changes in Kafka topology, Zookeeper needs to update the overall Kafka configuration. Kafka is broadly used by organizations in some combination of virtual private clouds say 34%, public clouds 52% and on premises 57%. Half of these organizations are using Kafka for microservices where microservices send messages or streams to Kafka and the target microservices can read the messages or stream data. Moreover, Kafka can be used as a log monitoring tool. Kafka can read log files locations and send logs to a centralized location through Kafka streams. It's ability to send data in bulk, high throughput and scalability has made it a reasonably good choice for many enterprise applications. Currently, there are some tools e.g. LinkedIn's Burrow and Datadog being used to monitor Kafka's performance by monitoring Kafka consumers. [7]

2.2.2 RabbitMQ

RabbitMQ is an opensource message communication mechanism which uses Erlang based Advanced Message Queuing protocol. It can support clustering and other complex messaging mechanisms. It was first launched in 2007, however, its stable version was released in 2015. Erlang is very common in telecommunication based systems due to it's higher availability as only 32 millisecond downtime per year has been reported. RabbitMQ uses Erlang based database to store data. It is in-file data storage which stores users, exchange, bindings etc. The messages along with their status (whether delivered or not) are stored in the queue. RabbitMQ supports external plugins, for example, an administration control plugin can be added to the

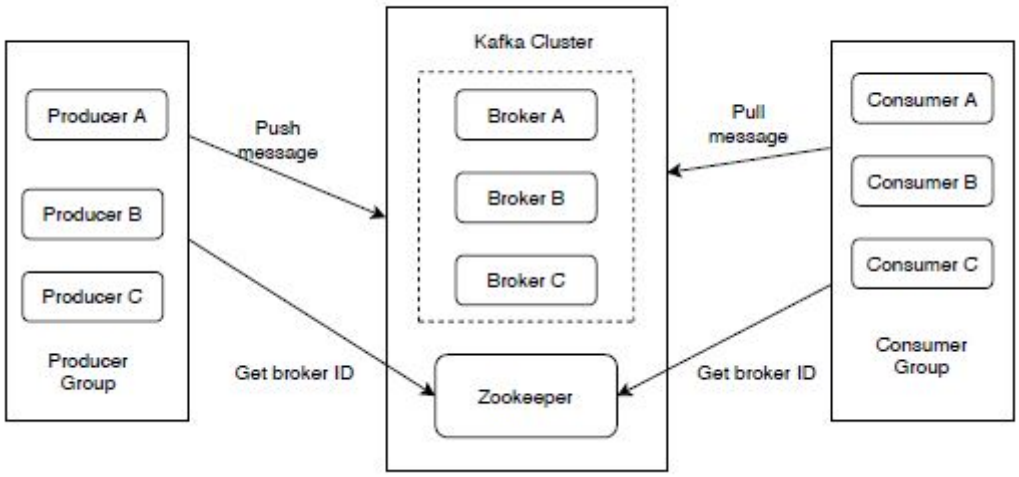

Figure 2.2 *Kafka Basic Architecture*

RabbitMQ for monitoring purpose. It can also be deployed as a single instance or in a cluster.[14]

AMQP which is one of the protocols used by RabbitMQ, is an open standard protocol used for messages exchange. It represents the messages format and commands to be exchanged. The protocol provides broker and client implementation in various programming languages. AMQ protocol constitutes the following concepts:
- It acts as a message broker. Takes messages from publishers and passes to consumers.
- Use Virtual Host mechanism to separate the applications which happen to use the same AMQ instance.
- Uses a TCP network connection between the broker and the application. A channel or virtual connection residing inside the connection for authorization when messages are published and consumed.
- Exchange applies the routing rules to the messages. The routing rules include source-destination, topic (publish-subscribe), multi-cast and header exchanges.
- A queue is a sequence of items stored inside the broker.
- A binding which allows flow of messages from an exchange to queue. [13] RabbitMQ uses Round Robbin queuing mechanism and removes the message from its database once it is delivered. Some of the key features of RabbitMQ are: - Multiple local servers can be grouped together to form one broker to handle with fault tolerance and load bearing. - RabbitMQ supports multi-platform connections e.g. MSMQ protocol used by C# is accepted by RabbitMQ using AMQP protocol.[14]

2.2.3 ActiveMQ

ActiveMQ (written in Java and implements JSM standard) is also a message broker and follows the same message delivery mechanism as other message brokers do. Many ActiveMQ brokers can be combined to create a broker network which increases the performance capabilities. On contrary with RabbitMQ, without a middle-ware ActiveMQ boker cannot send data from a C# based application using MSMQ to a Ruby based application using STOM due to the limitation that JSM based applications can only directly transfer data between Java platforms only. [14]

2.3 Microservices Monitoring, Logging and Tracing

Old data can be very useful at times to predict the future events. In the past, amazon and Delta Airlines had to go through immense loss due to the unavailability of system's internal data as they couldn't predict the failure in their data-center which caused the systems shut down. Similarly, in case of microservices we need to keep track of some information such as CPU usage, memory usage, request initiator, request latency time to keep a check on microservice's health and working ability. Keeping a track of internal states of microservices (or systems in general) is called observability and generally it can be achieved by logging, metrics and tracing. Here, logging means tracking the sequence of events.

In logs, everything about an event such as timestamp, latency, event's trigger source and status of the event/call can be written. The logs are stored as later useful information can be deduced from them. Metrics is the measure of event's attributes. E.g. number of events occurring per minute is a metric. Similarly the latency time or the average of latency time of all the events per minute can help determine the response or execution time. These metrics can be very helpful in monitoring the system. For example, you can monitor the system by configuring an alerts which triggers if the latency time is below a certain threshold. Logs can also help perform tracing. Tracing is about tracking a sequence of events within a system/microservice or within multiple systems/microservices. Within the same system it is possible to trace the request to identify where exactly the delays have been occurred. Similarly, events can be traced from one microservice to another to identify the failures or delays. [20]

2.3.1 Monitoring

Managing a system of several microservices is quite challenging. A lot of aspects need to be taken care of to keep the system healthy and running. Every microservice produces it's own logs, the latency problems can occur anywhere and different performance behaviours are required from various microservices. Without a proper

centralized monitoring tool it is quite tedious to take care of all these individual aspects by logging into the microservices separately and scanning the right information. Hence, it is important to have a bigger picture by combing the smaller chunks together. To achieve this, one needs to start with the simplest picture which is starting to monitor by a single node or server. Three different scenarios of monitoring have been described below with monitoring level increasing from simpler to complex. [21]

Single Node and Single Server Monitoring: A single service running on a single server requires the monitoring of the host, the server logs and the performance of the microservice itself. Monitoring the host might involve monitoring the CPU and the memory. Server logs can be monitored using some monitoring tool. The most important task will be monitoring response/latency time or the error rate of the microservice. To obtain the information on latency time/error rate, one can monitor the logs of the server hosting the microservice or directly the logs coming out of the microservice.

Single Node and Multi Server Monitoring: In this scenario multiple instances of the same service might be running on different hosts. A load-balancer can be used to manage these instances. To monitor this kind of setup, one needs to monitor the host to get metrics such as CPU and memory and the logs of the microservice instances. A monitoring system Nagios can be used to achieve this collective monitoring of the logs of microservice instances. Response times could also be monitored by tracking the traces from load-balancer. In case an issue is detected, it is advisable to debug both the load-balancer logs as well as the microservice logs.

Multiple Services Running on Multiple Instances: In this scenario several services are running on multiple hosts to provide the desired functionality to the users and each service can have more than one instances. Detecting problems and debugging the issues in this case becomes the most tedious as developers need to ssh into several instances to scan the logs and find the problems. The most efficient way to cater to this is by aggregating the logs and metrics in one place. The below diagram depicts the kind of a system running multiple services on multiple hosts.

Figure 2.2 shows a small system of microservices where each service is hosted on a different host/instance.

Microservices reside independently from each other and written in multi programming languages and can be deployed on different platforms, hence, their constant monitoring is required to make sure they do not break before and after a particular release and whether their overall performance meets the expectations. [16] Microservices based systems are designed to be fault tolerant, hence, they should be able to handle the failure due to unavailability of the required infrastructure. Negligence in this area can cause a bad end-customer experience. The performance

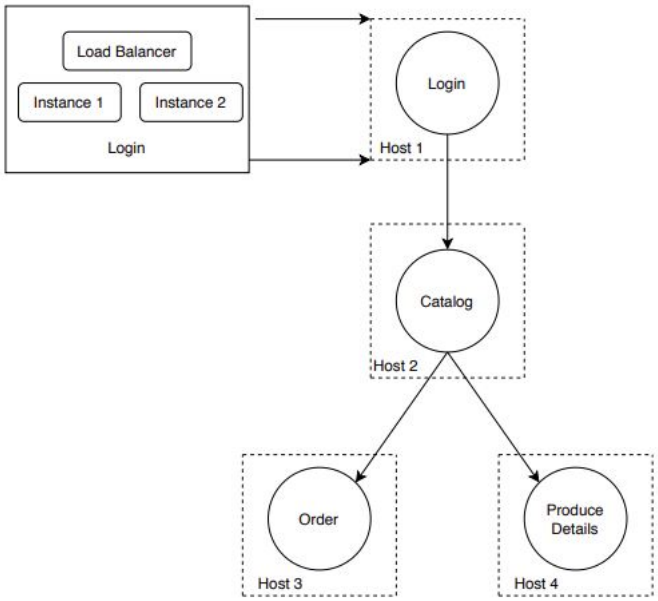

Figure 2.3 *Microservices deployed across a distributed system*

metrics are usually the latency time, CPU utilization, services health status, memory, disk usage and errors.

Additionally, architectural elements such as number of calls per minute and business metrics should also be continuously monitored with some monitoring tool. At minimal, a monitoring tool displaying status of metrics such as latency time, throughout in a dashboard is needed to meet the expectations of development teams to overcome the challenges related to services communication. [10] [16]

In monolithic systems, the system metrics such as CPU, disk IO, memory and network utilization are tested using black box techniques. In white box metrics monitoring (which is a low-level monitoring), system and database logs are generated and sent to the metrics monitoring tool through SNMP. However, as the deployment patterns of microservices might vary from service to service and services might use additional message exchange brokers, using a time series database is better as it supports data labelling and real time querying. [11]

Apart from black box and white box monitoring, there are various commercial and opensource microservices monitoring tools available. Some tools perform component level monitoring and some perform domain specific monitoring. And the monitored metrics can also be visualized in graphs using a metrics monitoring visualization tool.[15]

In the subsequent subsections a few microservices monitoring tools have been described.

Prometheus

Prometheus is a microservices monitoring tool which was developed by SoundCloud. It uses a time series database and LevelDB as a storage engine. Prometheus can be integrated with other monitoring tools such as Grafana to view time series data. LevelDB supports querying the data from outside via an HTTP API. To get the metrics inside Prometheus the microservices need to be instrumented with Prometheus exporters. There are two ways to instrument. First, standalone exporters can export metrics on http by consuming microservices logs, secondly, exporters can be instrumented inside the microservices using programming language specific libraries. Currently, there are libraries available in various languages, for example, C#, Python, Java, NodeJs and PHP. Some third-party systems can also be used to export Prometheus metrics. Examples of these third-party systems are databases such as PostgreSQL, MySQL, MongoDB and Oracle. Also hardware, messaging systems (e.g. Kafka, RabbitMQ), storage, HTTP and APIS. It can also collect data from systems like Kubernetes, CollectID, NetData and Grafana which already expose metrics in Prometheus format. [11]

AppDynamics

Appdynamics in a commercial tool which can be used for application performance monitoring. It works for both developers and architects and can be used for testing in developmental environment as well as production environments. Appdynamics thoroughly records system performance metrics by tracing all the requests of a transaction. It can send out the alerts if the system exhibits an unusual behaviour.[16]

Appdynamics is capable of collecting performance and business metrics of your application. It can also help you track the requests to the particular code line which can save a lot of time a developer needs to spend on debugging. It is possible to add watchpoints to the code, these watchpoints will gather the data about errors, stacktraces and other critical information. Watchpoints are non-stopping and can be added to the application without having to restart or stop it. Appdyanics UI provides charts, dashboards to view the transaction, performance and container network metrics. It also provides an overall map of the whole network indicating nodes, transaction flow and node health.[17]

New Relic

New Relic is an application performance monitoring tool which has been registered with around 14000 companies. It collects real time application performance data to help create insights on application performance, problems and solutions. The tool is capable of providing various features such as APM, Infrastructure, browser and

insights etc. The APM feature helps users to see the charts with deep real data which depict performance bottlenecks of application, server or a database. New Relic infrastructure feature provides system properties such as memory, CPU usage and can be configured to send alerts in critical situation. Browser feature can help identify and improve page load time and resolve front-end errors. New Relic insights can help visualize the data collected through metrics explorers into different charts. Looking at these charts viewers can analyze the system's performance. [18]

Grafana

Grafana is a visualization tool which can be used to monitor bulk of data in the form of graphs, charts or tables in order to extract some useful information out of it. These graphs can display multiple queries along the series, use different units for data and colors for the graphs. Grafana allows graphical visualization of bulk of data tracked over time. [19] For example, IoT sensor devices produce a lot of data which gets stored in the cloud environments. However, without a visualization tool this data is meaningless. Grafana is one of the most commonly used open source data monitoring tools which is available as a web application and displays different graphs, charts or tables in the form of panels on a dashboard. Grafana allows connectivity to various open source applications and time series databases e.g. InfluxDB and Elasticsearch. IoT and real time data can be effectively stored in these time series databases. Although external applications can be used with Grafana as plugins, nevertheless, Grafana has a very tight relationship with time series database. All the visual results on dashboard e.g. graphs and charts are a result of query to the time series database. Grafana functionality to fetch and display data in form of graphs depends on the functions it's time series database can provide which poses limitations for some users. Limitations are identified as [8]

- Inability to perform customized query. • Certain applications cannot be plugged in with Grafana due to their inability to conform with time series database functionality.

2.3.2 Logging

Logs are very crucial in a way that you can detect the errors, stacktraces, other information about services calls and data flows by looking at them. In case of a monolith application where a single instance is running, it is easy to ssh into the machine and look at the logs from the log file to find out if any problematic thing is going on. However, the same task is very complicated in case your system is comprised of many microservices. The solution to this problem is to gather all the logs from different microservices to one place. Another important feature should be

the ability to apply some query to analyze logs from a specific microservice, host or a process ID to make it easier to find what the person is looking for. [19]

ELK: Elasticsearch, Logstash, Kibana

ELK is an opensource tool which consists of Elasticsearch, logstash and Kibana. It has the ability to collect, aggregate and display the logs in textual as well as graphical format. Kibana is a dashboard where you can search the specific logs using a query or see logs in a graphical form such as charts and graphs. Elastic search is a document database engine which is used to store the log data. It provides restful APIs, client libraries and supports the query application on the stored data. Logstash converts log data from various formats to one single consistent format. It can read logs from Nginx, Apache server and parse syslogs. It reads log data in form of JSON over UDP protocol. Source libraries in various languages are available to perform this task of sending data to the logstash. Developers can define logging levels for almost all the libraries. The common logging levels are debug, info, warn, error and fatal with debug the least and fatal the most serious. It is also possible to define a threshold on the logging levels if displaying all the logs is not required. For example, the threshold on warn will only send logs with level warn, error or fatal. Hence, developers can enable lower level logging for the development environments where they need detailed information and a higher logging level for production environments.[19]

In a system of microservices logs from different microservices are collected, fetched using Logstash and stored on ELK stack server where they can be visualized in Kibana by developers. These logs provide lower level details, hence, debugging practices by analysing logs can be used to detect interaction problems within microservices. Logging can be categorized into three levels. In basic logging level, which is like traditional monolithic logging, developers must check and analyse microservices logs manually in case an issue or failure is encountered. These logs can include various kinds of useful information e.g. time when a log was generated, executed method and context information. The second logging level is based on visualization. In this method logs are picked based on conditions and regular expression and the selected set of logs can be examined to study the microservices execution behaviour by creating some statistical charts. [3] In ELK stack all this debugging and data visualization can be conveniently performed in the Kibana UI. Once the logs are fetched via logstash, user can log into Grafana web interface where data can be visualized in different graphs and charts after applying some ad-hoc queries on it. Queries are also very effective at filtering the desired data in the pool of logs. [19]

2.3.3 Tracing

Distributed tracing is a mechanism which allows performing insights on individual microservices and the system as a whole. It is used to monitor the performance, debugging the systems, monitoring logs, identifying root causes and failures. Distributed tracing uses the logs correlation in which certain patterns or sequences in the logs are identified and analysis is performed to make certain predictions. [21]

Google Dapper System

Dapper is a monitoring tool which was developed for Google's production distributing system. Dapper started as a tracing tool but evolved as a monitoring platform. It was initially developed to trace Google's internal system of microservices. Dapper's tracing logic is like the previous tracing tools e.g. Magpie and X-Trace, however, it was designed to have better sampling and instrumentation logic. Dapper system outperformed in Google's environment when tested against similar tracing systems for over two years. It causes lower overhead on the performance of microservices system, tracing instrumentation is kept minimal, system can scale with respect to the data size and processes information efficiently to display results quickly in the monitoring tool. When a system needs to send tracing information it requires to be instrumented to generate a trace. In Dapper, a trace can comprise several spans where a span is a log of time stamped record which includes the span's start and end time. A span can have a span name, parent id and a span id except the parent level span which does not include parent id. All spans under a specific trace must share the same traceID. Once a request is completed, microservices call path is determined based on spans sharing a common trace ID. Dapper's trace logging and collection comprises three phases. First of all, the tracing data is written to logs then data is fetched from all hosts via Dapper demons and stored in Dapper's Bigtable repository from where it can be fetched to display for monitoring purposes. Dapper's API can be used to fetch this tracing data to build the analysis tools. [4] Open Source microservices monitoring tools like Zipkin [5] and Jaeger [6] are based on Google's dapper technology and are widely being used by companies using microservices architecture to monitor calls and services dependency networks.

Spring Cloud Sleuth

Spring Cloud Sleuth performs distributed tracing for spring microservices. It borrows some concepts and solutions from Google's Dapper System. Sleuth creates spans which include the work between two points in a request. For example, if a request is made from order service to the product_purchase, Sleuth generates three spans e.g. one for the order internal work, second for order to product_purchase

and third span for the product_purchase internal work. Each of these spans contains a unique ID. Spans contain a parent ID except the parent span so that all the related spans could be grouped. A trace is a collection of related spans. These spans are stored in the logs, hence, requests can be tracked or filtered by applying search operations on the aggregated logs. To work with the Sleuth, a dependency needs to be added to the code. Sleuth will take care of tracing the requests and adding this information to the logs. For an incoming request, the tracing information is extracted by Sleuth and stored in the logs. Similarly, sleuth attaches the tracing information to the outgoing requests to make it available for other microservices. Spring cloud Sleuth can also be configured to send other information such as service name along with span and trace IDs to the logs. This particularly helps in tracing the calls in a system of multiple microservices. It is also possible to configure Sleuth to allow sharing logs and traces with the Zipkin server. [20]

Zipkin

Zipkin is an opensource distributed tracing tool. It helps in tracing the call sequences from one microservice to the others. Zipkin also gathers the timing data for calls which helps in monitoring system events. Zipkin has four main components which are collector, storage, search and web user interface. The instrumented applications generate and send the tracing information to the collector component of Zipkin which then validates, stores and index data in the storage component. Zipkin supports Cassandra, Elasticsearch and MySQL for data storage. The search component is comprised of a JSON API which fetches the indexed data from the storage and displays it on the web user interface. The Web component of Zipkin has two main features. The first features allows filtering the calls/requests data (with end points and latency time) using filters such as 'Service Name', 'Request Duration' etc. The other feature displays a call dependency diagram which is very helpful in monitoring the system which has several microservices communicating with each other. However, to send data to the Zipkin server, the code needs to be instrumented with the relevant libraries. On the other hand, setting up the Zipkin server is quite easy. It can be run inside a docker container by executing a simple docker command or can be installed onto the server of choice. Zipkin's user interface default port is 9411. Zipkin can also receive metrics from the Sleuth for spring boot projects to obtain logs and other tracing information. Once tracing data sharing is enabled within Sleuth, it shares the metrics and traces in Zipkin readable format.[20]

Jaeger

Jaeger is another opensource distributed tracing system. Jaeger borrows the same inspiration and concepts as Zipkin and Google's Dapper system. Jaeger was developed by Uber to monitor their distributed system. Jaeger uses opentracing library which is available for various programming languages. To monitor with Jaeger, the source application needs to be instrumented with the relevant opentracing programming language in order to send tracing information to the Jaeger. Once Jaeger is setup, the call requests with latency time can be viewed in Jaeger user interface (UI) which is by default available on 16686. Jaeger also displays a dyanic call dependency graph of the system of microservices. Jaeger can be easily setup inside a docker container by running a docker command.[20]

3 Opentracing Tools Comparison

This chapter draws a comparison between some of the opensource and commercial microservices monitoring and tracing tools. Chapter covers the complete process of how the list of tools was prepared, data extraction for each tool and lastly the features comparison in tabular form.

3.1 Tools Selection

Various microservices monitoring and tracing tools were taken into account to get information about their features, installation, compatibility with other tools and their usage in industry. To get the list of tool names below search query was executed in some popular search engines such as google scholars, IEEE and university databases:

("open tracing" OR opentracing) AND tool*

Results which did not mention anything significant about the tool either in title or abstract were excluded. The results were mostly links to the tools official websites, tutorials, documentation and blogs. During the final step, tools which didn't have any official website or had lack of documentation were excluded from the final list of tools to be consider for comparison. Below is the list of tools which was selected for comparison:

- AppDynamics
- Datadog
- Elastic APM
- Jaeger
- InspectIT
- Instana
- LightStep
- SkyWalking
- Stagemonitor
- Wavefront VMware
- Zipkin

3.2 Data Extraction

Data regarding tool's features, installation and usage documentation was collected from the official websites of tools. Each tool was individually explored for its characteristics and features. The result/information was noted down the paper. Once the initial data collection process finished, a table was created which only comprised the features which were common among most of the tools. The final features on which the comparison was made are:

- APIs Availability
- License
- Programming Language
- Installation
- Supported Back-end/DB
- Setup Requirement
- Support for external Metrics/tools
- Support for external tracing tools
- Support for external visualization tool
- End User Monitoring
- Kubernetes Monitoring
- Container Monitoring
- Context Monitoring
- Agent Enable/Disable option
- Data Retention
- Dashboards
- Charts/Graphs
- Application logs
- Query/Tag filter
- Alerts
- Metrics

3.3 Comparison

In this section three comparison tables have been presented which compare around eleven monitoring and tracing tools. Eleven tools have been divided into Table 3.2 and Table 3.3 which compare the data relating to tool's installation, operations and additional functionalities (such as integration to an external back-end, metrics monitoring or tracing tool/data storage). Whereas, the Table 3.1 compares the performance metrics such as timing data/latency time, call count, error rate, page load time, CPU usage, memory usage and dependency diagram's availability.

Table 3.1 *Tool Comparison - Metric*

Metric	Zipkin	Jaeger	Light Step	Instana	Sky Walking	Inspect IT	Stage Monitor	Datadog	Wavefront VMware	App Dynamics	Elastic APM
Timing data for requests	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
Call Count	yes	yes		yes							
Error Rate			yes	yes			yes	yes	yes	yes	yes
Page Load time				yes			yes	yes		yes	yes
CPU Usage			yes (by instrumenting)	yes	yes	yes	yes	yes	yes	yes	yes
Memory Usage			yes (by instrumenting)	yes	yes	yes		yes	yes	yes	yes
Dependency Diagram	yes	yes	yes	yes	yes	yes		yes	yes	yes	yes

Table 3.2 *Tool Comparison - Features about the tool itself (independent of usage)*

Feature	Zipkin	Jaeger	Light Step	Instana	Sky Walking	InspectIT
APIs Availability	yes	yes	yes	yes	yes (POST APIs)	yes
License	Apache License 2.0	Apache License 2.0	Community (free), Pro (100$ /month), Enterprise (custom pricing)	75euro /host/ month Billed annually	Apache License 2.0	
Dependency Diagram	yes	yes	yes	yes	yes	yes
Alerts	no	no	yes	yes	yes	yes
Charts/ Graphs	no	no	yes	no	no	no
Agent on/off	yes	yes	no	yes	yes	yes
Installation	manual instrument	manual instrument	auto instrument	install agent	auto+ manual instrument	attach agent to JVM
Programming languages	C#, Go, Java, Java script, Ruby, Scala, Php + community supported libraries	Client libraries for Go, Java, Python, NodeJs, C++, C#	Java, NodeJS, Go, PHP, javascript, python, Ruby, C++, C#	16 infrastructures, 21 tracing libraries/ langs, 12 alerting platforms, Jenkins/ Mina CI/CD, 5 Logging systems	”Java, .Net Core, PHP, NodeJS, Golang, LUA agents. Support for Istio + Envoy”	”JRE 1.8.0 + Including Oracle, JDK and Azul”

Table 3.3 *Tool Comparison - Features about the tool itself (indipendent of usage)*

Feature	Stage Monitor	Datadog	Wavefront VMware	AppDynamics	Elastic APM
APIs Availability	no	yes	yes	yes	yes(APIs as methods on client class)
License	Apache License 2.0	Various monthly pricing plans	Approx. 15$ per host per month (charged annually)	APM pro (free trial) APM advanced (request pricing) APM peak (request pricing)	Apache License 2.0, Pay for backend resources
Dependency Diagram	yes	yes	yes	yes	yes
Alerts	yes	yes	yes	yes	yes
Charts/ Graphs	yes	no	yes	no	no
Agent on/off	yes	yes	yes	yes	yes
Installation	Integrate plugin	Agent + manual instrumentation	instrument with SDKs	install agent	instrument code with agent
Programming languages	Java	”Official client libraries Scala, Python, Java, NodeJs, Rust, Ruby, PHP, Go, Elixi, .NET Non-official: Scala, Rust”	all common Spring Boot components	”Java, .NET, Node.js, PHP, Python, C/C++ and more. Supports iOS and android apps instrumentation. Setup browser Real User Monitoring”	Java, Go, Node.js, Python, Ruby, .NET and Javascript

4 The MsViz Tool

This chapter describes the overview of MsViz tool (which is a final product of this thesis), the components it is comprised of and the metrics related to microservices which can be visualized using this tool.

4.1 Purpose, Overview

As mentioned earlier in the background section, there are various opensource and commercial microservices monitoring and tracing tools available which can provide information about overall health and status for a system comprised of several microservices. Also, comparison of some opentracing tools have been drawn in the previous section from which it is evident that there is no single tool which can provide all the necessary insights related to performance and business metrics inside one tool. Some of the tools which provide more features than others are the commercial tools and might cost a lot to the company on monthly or yearly basis in order to provide performance and business monitoring onto their microservices based systems. As per the literature done during this thesis, there are certain opensource monitoring and tracing tools which are widely being used by many software companies such as Prometheus (as a monitoring tool), Zipkin and Jaeger (as monitoring and tracing tools). Besides these tools, Grafana is used to visualize system metrics in form of graphs and charts to make quick and subtle judgements onto the system's overall health and condition. However, to perform a thorough monitoring, all these tools need to be separately setup and constantly monitored. Furthermore, these tools lack the insights about business metrics which can be very important to the companies to make strategic decisions for the future of their system's individual components/services. Hence, a tool which could provide some important metrics and tracing information in one place along with business insights can overcome this gap.

MsViz (a microservices metrics visualization tool) provides a solution which combines some of the important metrics and tracing data from Prometheus and Jaeger server combined together with business metrics stored in a mySQL database and creates a visualization diagram which provides all these metrics in one place. It draws a basic call dependency graph of a system of micro-services and displays certain performance and business metrics for each microservice inside the call graph. The tool is comprised of four Grafana plugins. Two of which (a MS-Visualization-Panel plugin and jaeger-backend-datasource datasource plugin) were developed as a result of this thesis.

4.2 Tool Description

The MsViz tool is comprised of three datasources and one front-end panel plugin. The tool utilizes the backend/datasource plugins to gather all the required metrics data from the relevant monitoring and tracing tools servers then draws a diagram with the available information inside front-end plugin. In order to use these plugins, it is assumed that the microservices of the system to be monitored have been instrumented to send required metrics to the Jaeger and Prometheus server. And a MySQL database contains the business metrics to be displayed inside the final output of tool.

The tool is comprised of build-in and custom plugins which include Grafana's built-in Prometheus datasource plugin (to obtain performance metrics from Prometheus server) and MySQL plugin (to obtain business metrics stored in an external MySQL database). Third datasource plugin named "Jaeger-backend-datasource" was created to acquire microservices call dependency and call count data from Jaeger server. To use these plugins, URLs to the respective servers need to be entered to the plugins configuration user interfaces inside Grafana web server. Jaeger-backend-datasource fetches data from Jaeger server and converts this data into a format which is readable by Grafana internal backend. Front-end plugin named "MS-Visualization-Panel" automatically fetches data from datasource plugins and plots a dynamic call dependency graph with performance and business metrics displayed inside. Figure 4.1 describes how the various plugins and tools have been combined together to form the MsViz tool.

4.3 Output Graph Metrics

Apart from requests/calls tracing data to create parent-child relationship, the MsViz tool displays metrics which can be categorized as performance and business metrics.

4.3.1 Performance Metrics

Performance metrics include two different metrics e.g. latency time or service response time and load time. Response time is the average time of all the responses in last minute which a particular microservice took. For example, if a microservice processed three requests in last minute, where each response took 0.7ms, 0.8ms and 0.9ms respectively then average response time will be (0.7+0.8+0.9)/3 = 0.8ms.

Whereas load time indicates the number of requests a micro service processed during the last minute.

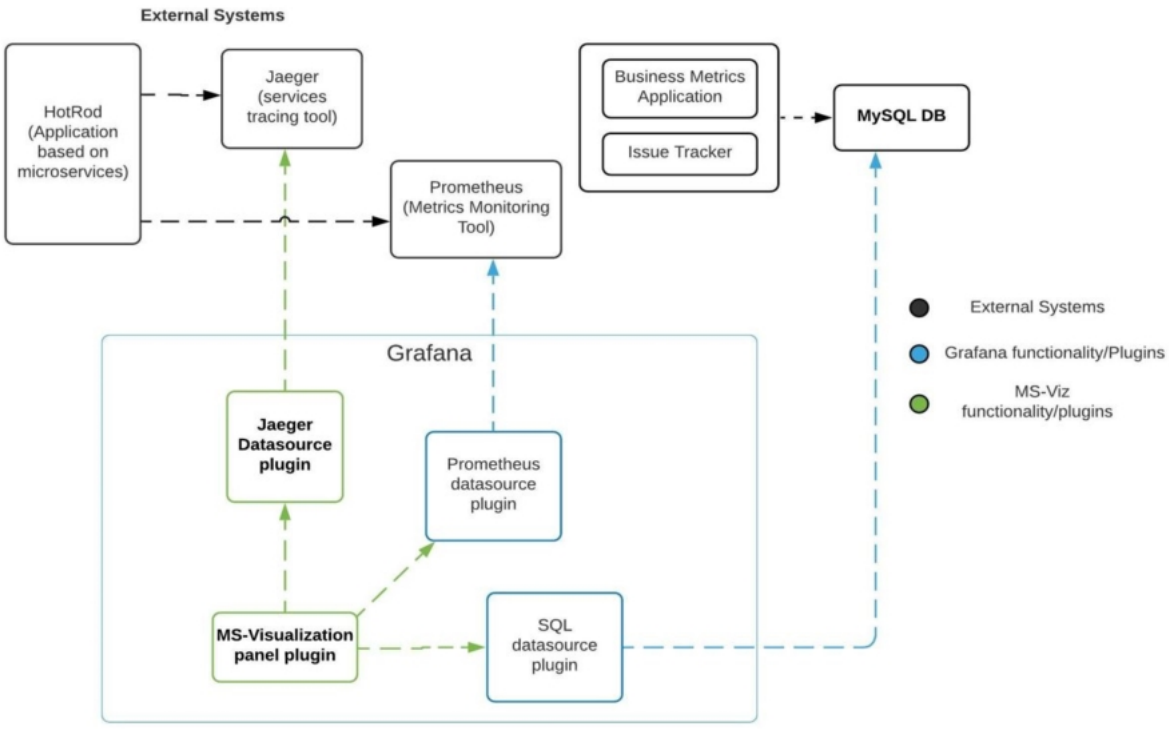

Figure 4.1 *MsViz Tool Components*

4.3.2 Business Metrics

Business metrics can be divided into three categories, such as:

- **I**ssues: Include the number of open and closed issues against a particular microservice. A comparison is drawn inside the graph between these two counts.
- **B**ugs: Include the number of open and closed bugs against a particular microservice. A comparison is drawn inside the graph between these two counts.
- **M**onetary Metrics: Includes the revenue (without a unit of currency), cost (without a unit of currency) and total effort (without a unit of time, can be assumed as hours) spent against a particular microservice. A comparison is drawn between revenue and cost whereas effort is shown as a separate metric inside the graph.

5 MsViz Implementation

This chapter describes the tool from implementation perspective. In first two sections, the tools which have been researched and selected to gather data are described, in subsequent sections the tool selected as a baseline of MsViz is described later implementation of self developed Grafana plugins is explained. Then built-in Grafana datasources are described in regards with their usage in MsViz tool. Lastly, the results of the MsViz tool and the testing process is described.

5.1 Tools Research

During the literature review on relevant material, it turned out that there was not much academic research available on microservices metrics visualization tools and techniques other than Google's Dapper technology which is mainly being used by Jaeger and Zipkin.

Initially the research also considered investigating the possible ways to create microservices dependency graph without manually instrumenting the code (adding code to generate the tracing data in order to send to the monitoring/tracing tool's backend/server). To research this possibility, the logic and working of open source microservices monitoring and tracing tools e.g. jaeger and zipkin was studied and turned out that developers need to instrument their code with opentracing libraries in order to send traces to the Zipin or Jaeger server to be processed to display the results. Hence, other possible ways were investigated to determine the possibility to generate the microservices call dependency graph without manually instrumenting the code.

According to the literature review of microservices message based communication protocol, it came into knowledge that Kafka is one of the most popular message brokers which is used for asynchronous communication between microservices. Kafka messages format have been studied to see if the information about the request sender and receiver microservices could be extracted. The research findings showed that Kafka is pretty agnostic to the messages. Producer groups produce messages which do not contain the information about the calling or called microservices. These messages are directed towards Kafka topics. The messages are basically comprised of Key, Value. On the other hand, consumers constantly check Kafka topics assigned to them to look for the messages with relevant key. Multiple producer and consumer groups might be sending and consuming messages from the same topic. Also, multiple microservices might be using similar producer or consumer groups to produce and consume messages. Considering this information, it was not possible to get the

calling and called services names by monitoring the Kafka message bus in a network of microservices. However, it is possible to add custom headers (with services names/IPs) to the messages before they are directed to a Kafka topic by a producer group. The information about calling and called services can then be extracted when a relevant message is found by Kafka consumer and is ready to be consumed. Nevertheless, this approach is not very suitable as it requires instrumenting the Kafka messages with proper headers and it can cover only for microservices communicating through Kafka message bus. Also, no previous research has included any study around microservices visualization techniques without having to instrument the code using Kafka or other message broker technologies.

Additionally, some logs monitoring tools e.g. Elastic stack and event monitoring tools such as Prometheus were studied as part of this research to assess if a dependency graph could be created by monitoring their events/requests data. Prometheus is an event monitoring and alerting tool and uses time series database. Prometheus metrics were analyzed and there was no information about the services call pattern. The information in Prometheus is generally about resources being consumed e.g. UP services, counts of errors, CPU/memory utilization, requests count, tasks etc. On the other hand, Elastic stack is a centralized logging tool. Elastic stack indexes the logs in Elasticsearch via Logstash and Kibana board displays the logs from all the services configured to send their logs to Elastic stack through logstash. Elastic stack can provide a way to figure out called and calling services if the logging level defined in application send enough information e.g. microservices names in a message before a call to external service is made which is something developers need to manually do inside the code while generating the logs. Hence it was not a possible option either for microservices monitoring with no instrumentation goal.

Jaeger and Zipkin are two of the most popular opensource tools to monitor network of microservices and are being used by many microservices based companies. However, they do provide a very basic dependency diagram which only depicts the relation from calling to called microservies and the total calls count from one microservice to the other.

5.2 Tools Selection

After performing research on some opensource monitoring and tracing tools, the following tools were selected to construct the MsViz tool.

- **J**aeger: Microservices calls dependency data (in form of parent-child relationship can be obtained from Jaeger.)
- **P**rometheus: Prometheus is selected to get the performance metrics such as requests latency time and requests load on a microservice.

- **M**ySQL: MySQL is used to store business metrics such as open/close bugs, open/close issues, cost, revenue and effort spend on a microservice.
- **G**rafana: Grafana supports custom plugins for external data visualization. MsViz tool is developed in form of Grafana plugins.
- **D**3JS: To improve the existing call dependency graph from Jaeger tool, D3JS, which is a graph plotting library have been utilized to draw the desired dependency graph whereas the data has been fetched from Jaeger server by using Jaeger API to fetch the microservices dependency data.

Different icons are used to represent microservices, database and message buses. The graph also includes the information on the number of calls made from one microservice to another. The final graph output is embedded to the Grafana monitoring tool.

5.3 Grafana Plugins

Grafana supports custom plugins development besides providing plenty of useful built in datasources, panels and dashboards. The custom plugins can be of three types.

- **Datasource Plugin:** The datasource plugin can be developed in order to fetch the data to be visualized form an external database or external server through an API, then data is converted to the Grafana readable format so it could be visualized in Grafana panels.
- **Panel Plugin:** Panel plugins are needed to visualize data in a desirable form such as chart, graph or table. These panels can utilize data from a Grafana datasource or static data can be used inside a custom plugin.
- **App Plugin:** App plugin is a combination of datasource and panel plugin inside one package. App plugins allows creation of different pages where it is possible to add some documentation or guide for the users.

5.4 Jaeger-Backend-Datasource Plugin

Jaeger-backend-datasource is a custom plugin which has been created as part of MsViz tool. The plugin is based on the example backend datasource plugin provided by Grafana on its official website. The Jaeger backend datasource plugin have been implemented as the following steps provided in subsequent subsections.

5.4.1 Setting up Development Environment

In order to develop and later test the plugin, Grafana environment was setup up and configured on Ubuntu 18.04 as a service. Grafana can be installed and run by cloning the Grafana repository from Grfana official gitHub repository, by running it as a service or by running Grafana Docker image. In order to run Grafana from source, Grafana repository needs to be cloned from GitHub then following commands in `src/github.com/grafana/grafana` directory need to be run to build Grafana frontend and backend respectively:
yarn start
make run

5.4.2 Updating Grafana Configuration

Grafana needs to know the plugins directory in order to include them and display in Grafana web server. The plugin directory URL needs to be written in Grafana configuration file. The configuration file directory in Linux is by default located in: */usr/local/etc/grafana/grafana.ini*

In grafana.ini file, following line needs to be commented out or added if it doesn't exist already:
[paths]
plugins = "/path/to/grafana-plugins"

Once, the grafana.ini configuration file is update, the Grafana server is restarted so the changes are reflected.

5.4.3 Plugin Structure

The Grafana plugin is comprised of two main files:

- **Plugin.json**

  Once Grafana is looking for plugins inside plugin directory mentioned in configuration file, it takes each individual directory containing a plugin.json file as a plugin. The plugin.json file must contain some important information about the plugin such as plugin type, plugin name and plugin ID. Where plugin type can be datasource, panel or app, plugin name describes the functionality or use of datasource in a few words and plugin ID should be a unique identifier so the plugin can be identified uniquely inside Grafana. The contents of plugin.json have been shown in Figure 5.1.

```
{
  "id": "Jaeger-backend-datasource",
  "name": "Jaeger-backend-datasource",
  "type": "datasource",

  "partials": {
    "config": "public/app/plugins/datasource/simplejson/partials/config.html"
  },

  "metrics": true,
  "annotations": true,
  "backend": true,
  "alerting": true,
  "executable": "simple-json-plugin",

  "info": {
    "description": "Jaeger datasource to fetch dependencies",
    "author": {
      "name": "Fouzia",
      "url": ""
    },
    "logos": {
      "small": "img/simpleJson_logo.svg",
      "large": "img/simpleJson_logo.svg"
    },
    "links": [
      {
        "name": "Followed GitHub Template",
        "url": "https://github.com/grafana/simple-json-backend-datasource"
      },
      {
        "name": "MIT License",
        "url": "https://github.com/grafana/simple-json-backend-datasource/blob/master/LICENSE"
      }
    ],
    "version": "1.0.0",
    "updated": "2019-04-10"
  },

  "dependencies": {
    "grafanaVersion": "6.x.x",
    "plugins": []
  }
}
```

Figure 5.1 *Plugin.json Contents*

- **Module.ts** The plugin's Module.ts contains the entry point of the plugin file describing the plugin logic. The file should extend GrafanaPlugin to any of the objects as PanelPlugin, DataSourcePlugin, AppPlugin. In case Jaeger-backend-datasource plugin it extends to the object DataSource Plugin and in case of panel plugin it extends to the PanelPlugin object.

The datasource plugin implements the following URLs:
`QueryFunction` This URL takes the external backend server URL from the plugin configuration page and makes a query request to the Jaeger API server. The function in Figure 5.2 from src/datasource.ts file gets executed when a query request is sent to the Grfana Jaeger datasource.
`TestDataSourceFunction:` This function is used to send a query response to the

```
query(options) {
  console.log("printing query options");
  console.log(options);
  const query = this.buildQueryParameters(options);
  query.targets = query.targets.filter(t => !t.hide);

  if (query.targets.length <= 0) {
    return Promise.resolve({data: []});
  }

  return this.doTsdbRequest(query).then(handleTsdbResponse);
}
```

Figure 5.2 *Query Function*

```
testDatasource() {
  return this.doRequest({
    url: this.url + '/',
    method: 'GET',
  }).then(response => {
    if (response.status === 200) {
      console.log("Datasource working");
      return { status: "success", message: "Data source is working", title: "Success" };
    } else {
      return { status: "failed", message: "Data source is not working", title: "Error" };
    }
  }).catch(error => {
    return { status: "failed", message: "Data source is not working", title: "Error" };
  });
}
```

Figure 5.3 *TestDataSource Function*

external data server (URL of external server is fetched from the datasource configuration interface) when ”Save and Test” button is clicked. The response ”Datasource is working” is returned if external datasource is running and ”Datasource is not working” in case query response is not received as success. The testDataSource function implementation logic is shown in Figure 5.3.

The Jaeger datasource is comprised of two sides e.g. the frontend and the server side. The frontend is written in typescript whereas server side is written in golang programming language. When a query from the panel is sent to a Datasource plugin, the ’query’ function inside /src/datasource.ts is called, which calls the Grafana server query function inside /pkg/datasource.go by passing the query parameters. The ’query’ function in datasource.go shown in Figure 5.4 is responsible for making the actual API call to the external Jaeger server. The Jaeger API response is converted into the required a specific format and result is sent back to the datasource query function inside datasource.ts which then returns the response to the panel where data can be visualized as per the panel’s implementation logic. The Figure 5.5 describes how communication between frontend and backend plugin takes place.

Following Jaeger API is called to gather microservices call dependency data:

```
func (ds *JsonDatasource) Query(ctx context.Context, tsdbReq *datasource.DatasourceRequest)
     (*datasource.DatasourceResponse, error)
{
    ds.logger.Debug("Query","datasource",tsdbReq.Datasource.Name,"TimeRange",tsdbReq.TimeRange)

    //Prepares tsdbReq
    queryType, err := GetQueryType(tsdbReq)
    if err != nil {
        return nil, err
    }

    ds.logger.Debug("createRequest", "queryType", queryType)

    switch queryType {
    case "search":
        return ds.SearchQuery(ctx, tsdbReq)
    default:
        //To create a metric query
        return ds.MetricQuery(ctx, tsdbReq)
    }
}
```

Figure 5.4 *Backend Server Query Function*

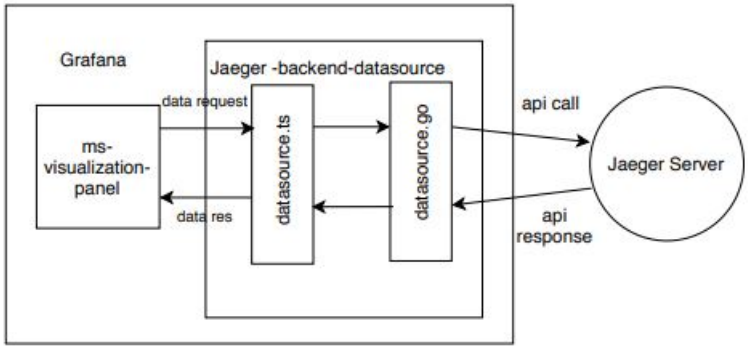

Figure 5.5 *Panel Plugin Request to Jaeger Datasource*

url-to-the-jaeger-server/api/dependencies/endTs=1605034853803&lookback=604800000

Where endTs and lookback are the optional parameters. Jaeger API returns the response in a format shown in Figure 5.6.


```
{
  "data":
   [
     {"parent":"frontend","child":"customer","callCount":1},
     {"parent":"customer","child":"mysql","callCount":1},
     {"parent":"driver","child":"redis","callCount":14},
     {"parent":"frontend","child":"driver","callCount":1},
     {"parent":"frontend","child":"route","callCount":10}
   ],
  "total":0,
  "limit":0,
  "offset":0,
  "errors":null
}
```


Figure 5.6 *Jaeger API Response Format*

```
{"results":
  {"A":
    {
      "refId":"A","series":[],"tables":
          [{
            "columns":[{"text":"Parent"},{"text":"Child"},
                       {"text":"CallCount"}
                      ],
            "rows":[ ["frontend","driver",1],["frontend","route",10],
                     ["frontend","customer",1],["customer","mysql",1],
                     ["driver","redis",14]
                   ]
          }],
      "dataframes":null
    }
  }
}
```


Figure 5.7 *Formatted Jaeger API Response*

```
To install frontend dependencies:
docker run -v ${PWD}:/opt/sjbd -w /opt/sjbd node:11 yarn install --pure-lockfile

To build frontend:
docker run -v ${PWD}:/opt/sjbd -w /opt/sjbd node:11 \
           node_modules/webpack/bin/webpack.js --config=./webpack/webpack.dev.conf.js

To Install backend dependencies:
docker run -v ${PWD}:/go/src/github.com/grafana/simple-json-backend-datasource -w \
           /go/src/github.com/grafana/simple-json-backend-datasource instrumentisto/dep ensure

To compile the backend:
docker run -v ${PWD}:/go/src/github.com/grafana/simple-json-backend-datasource -w \
           /go/src/github.com/grafana/simple-json-backend-datasource \
           golang go build -i -o ./dist/simple-json-plugin_linux_amd64 ./pkg
```

Figure 5.8 *Commands To Build Jaeger Backend Plugin*

Grafana queries needs responses either 'timeseries' or 'table' formats. Hence, the above Jaeger format is converted into the Grafana specific table format. The formatted response is shown in Figure 5.7.

5.4.4 Building the Plugin

Once, the datasource plugin logic is in place, the plugin's frontend and backend need to be built. The build creates a directory called 'dist' and keeps the compiled code in it. Later, when Grafana is restarted to reflect the changes, it looks for the plugin implementation logic inside 'dist' folder.
To build the Jaeger-backend-datasource plugin, the commands shown in Figure 5.8 need to be run inside datasource plugin repository:
Once, the plugins have been build, grafana server needs to be restarted to reflect the changes.

5.5 Prometheus Plugin

The existing Grafana Prometheus datasource plugin is used to call Prometheus APIs to gather performance and load metrics.
The following Prometheus API is called to fetch the performance metrics:
`GET/api/v1/query`

As Prometheus supports Pomsql query language, the queries to fetch response time and load time for each registered service is sent via Prometheus API. Then response is validated and converted to a suitable format.

5.6 MySQL Plugin

Grafana built-in MySQL plugin is used to fetch data from external MySQL database. The data in external MySQL database is comprised of open/close issues, open/close bugs, cost, revenue and effort for individual microservices. This data can be extracted from relevant project management and tracking tools and then stored in a table in MySQL database.

5.7 MS-Visualization-Panel Plugin

This section describes the implementation of MS-Visualization-Panel plugin which is a Grafana frontend plugin. The same pre-requisites as in subsection 5.4.1 and 5.4.2 were followed before starting the actual development.
The section describes how plugin integrates the data from all sources and generates the output.

5.7.1 Data Integration

MS-Visualization-Panel fetches data from all the datasources (Jaeger-backend-datasource, Prometheus and MySQL), then converts and integrates the data into a specific format in a single array. The plugin uses a D3JS graph plotting library which takes the formatted data as input and draws a microservices dynamic call dependency graph. D3JS graphically displays all the metrics fetched from datasources into a graphical form. The plugin has been developed in Typescript programming language.

In order to obtain data from Jaeger-backend-datasource plugin, the datasource is selected as 'default' datasource in Grafana datasources configuration interface. Hence, when user lands on MsViz frontend panel plugin, a data query request is send by default to the Jaeger backend datasource and the response data can be accessed from Typescript Props. The Figure 5.9 shows the code snipped which extracts data from Typescript Props.

```
constructor(props: Props){
    super(props);
}

async mergeMetricsData(){
    const { data } = this.props;
    console.log(data.series);
    if (data.series.length < 1) {
        console.log("No data received from Jaeger server.");
    }
    else {
            <Other logic here>
        }
}
```

Figure 5.9 *Jaeger Datasource Data in Props*

```
GET /api/datasources/name/datasource_name_here HTTP/1.1
Accept: application/json
Content-Type: application/json
Authorization: Bearer <API Key Token Here>
```

Figure 5.10 *Grafana Internal API Request Format*

To access Grafana's build-in datasources inside this panel plugin, Grafana internal datasource APIs are utilized. The following API has been used inside the panel plugin logic:
GET/api/datasources/name/:name

The request to the Grafana internal API has the format shown in Figure 5.10.

And the expected response contains the datasource Id as shown in the Figure 5.11.

Once the datasources IDs are known, MsViz frontend panel plugin requests data from Prometheus and MySQL datasources.
Front-end plugin contains two important files to implement the panel logic. The files are named "App.tsx" and "FormNetwrok.tsx". App.tsx is responsible for fetching data from all the datasources based on the input of a configuration file provided by the user. This configuration file contains the properties shown in the figure 5.12 and has to be placed in the default Grafana configuration directory (*/etc/grafana* in Linux based operating systems).
The configuration file is comprised of eighteen attributes, their values can be updated as required. The attribute 'api_key_admin' is an authentication token which is generated from Grafana API configuration user interface. The token needs to have admin rights for the MS-Visualization plugin to use grafana internal APIs.
Once data from datasources is collected inside App.tsx, 'mergeMetricsData' function calls 'getNetwrokData' function inside FormNetwrok.tsx file. getNetwork function

```
HTTP/1.1 200
Content-Type: application/json

{
  "id": 1,
  "orgId": 1,
  "name": "datasource_name",
  "type": "graphite",
  "typeLogoUrl": "",
  "access": "proxy",
  "url": "",
  "password": "",
  "user": "",
  "database": "",
  "basicAuth": false,
  "basicAuthUser": "",
  "basicAuthPassword": "",
  "withCredentials": false,
  "isDefault": false,
  "jsonData": {
    "graphiteType": "default",
    "graphiteVersion": "1.1"
  },
  "secureJsonFields": {},
  "version": 1,
  "readOnly": false
}
```

Figure 5.11 *Grafana Internal API Response Format*

```
{
    "grafana_url": "http://localhost:3000",
    "api_key_admin": "Grafana-generated-api-key-here",
    "services_status_query_prometheus": "up{job!='prometheus'}",
    "services_responseTime_query_prometheus": "performance_per_ms{job!='prometheus'}",
    "services_LoadPerMin_query_prometheus": "load_per_m{job!='prometheus'}",
    "mysql_db_table_name": "metrics",
    "erviceName_col_name": "service_name",
    "closed_bugs_count_col_name": "closed_bugs_count",
    "open_bugs_count_col_name": "open_bugs_count",
    "closed_issues_count_col_name": "closed_issues_count",
    "open_issues_count_col_name": "open_issues_count",
    "services_revenue_col_name": "revenue",
    "services_cost_col_name": "cost",
    "services_effort_col_name": "effort",
    "show_bugs_ratio": 1,
    "show_issues_ratio": 1,
    "show_costToRevenue_ratio": 0,
    "show_relative_effort": 1
}
```

Figure 5.12 *MS-Visualization-Panel Configuration*

Figure 5.13 *Typescript Data Array Format*

formats and aggregates all the data into one typescript aarry which is read by the D3JS graph plotting library. Once the data is merged and formatted it is returned back to the 'mergeMetricsData' function. The formatted data is stored in a typescript array in a format shown in Figure 5.13.

5.7.2 Output Graph

The function which plots the D3JS graph is called 'drawGraph' and is located inside 'App.tsx'. The function is called by 'mergeMetricsData' function with typescript data array as an input parameter. Finally, drawGraph plots the dynamic call dependency graph of microservices using D3JS graph plotting library and displays a legend alongside to describe the metrics data being presented inside the graph.

5.7.3 Building the Plugin

To build MS-Visualization plugin, the following command was run inside plugin directory:
yarn watch

The command compiles and builds the code inside dist directory. Grafana server needs to be restarted to incorporate the plugin changes.

5.8 Testing the Tool, Results

The tool has been installed and tested in Ubuntu 18.04. In order to test the tool in Ubuntu 18.04 server, following steps were executed step by step.

5.8.1 Prerequisites

The following steps were followed in order to run the tool.

- The Grafana server (version 6.6.0) was installed and run.
- Prometheus was installed and run.
- MySQL was installed and run.
- Docker was installed and run.
- Jenkins was installed and configured to run Grafana server, Prometheus, Jaeger and the test project.

5.8.2 Test Data Preparation

Jaeger "Hot R.O.D. - Rides on Demand" application has been used to test the MsViz tool. This is a demo application which has been developed by Jaeger to demonstrate the use of Opentracing API. The application is by default instrumented with opentracing library and sends data to the Jaeger server for monitoring. The application can be run by source or by running its docker image as per Jaeger Hot R.O.D instructions available on Jager Github repository.
To obtain the microservices performance and load, the application was instrumented with Prometheus to send the performance_per_ms and load_per_min metrics. The following business metrics were stored in a MySQL database named 'business_metrics'.

- Open Issues
- Closed Issues
- Open Bugs
- Closed Bugs
- Cost
- Revenue
- Effort

The business metrics do not reflect the actual data. In production environment, business metrics can be fetched from GitHub, project management tool or excel files (in case data is stored manually). For Hot R.O.D application, dummy business metrics are used to test the tool.

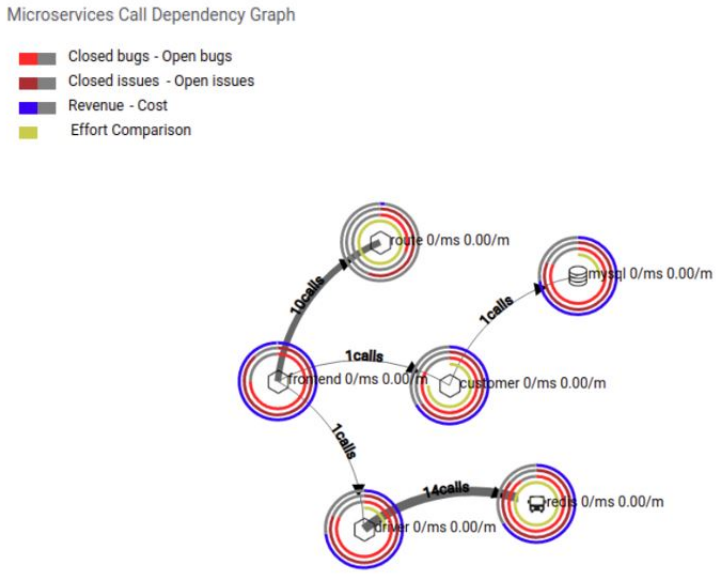

Figure 5.14 *MsViz Graph Output*

Lastly, the msvis.json file needs to be updated with the right properties and placed in Grafana's default configuration directory. (`/etc/grafana`). The Grafana configuration file was provided the directory URL of the pre-built plugins repository. Then Grafana server was restarted.

5.8.3 Running the Tool

In order to run and test, hot R.O.D application, all the datasources were configured in Grafana web server interface. The respective URLs of Jaeger and Prometheus were provided on Jaeger and Prometheus datasources. The MySQL datasource was configured by providing the MySQL host address aslocalhost:3306, database name as 'business_metrics' and a username and password for database authentication.
After adding the datasources, the output graph was viewed by accessing the 'MS-Visualization-Panel' from Grafana panels interface. Tracing data and values of all the metrics were successfully crossed verified for their validity. However, only the the positive use cases (with complete input data and with no input data) were tested. Negative test cases could not be conducted due to shortage of time.
The final output of the tool can be viewed in Figure 5.14.

6 Discussion

This section discusses the uncertainty factors and the limitations that can be predicted about this thesis work.

6.1 Uncertainty Factors

As the MsViz tool have been developed as part of an existing Grafana tool and it's functionalities, it is highly dependent on some external factors. For example, the tool might not work without proper maintenance with a higher or lower Grafana version than v6.6.0. Moreover, if the format of Grafana query requests to datasources or the Grafana internal datasource APIs change, this would need the MsViz corresponding code to be updated to ensure the tool remains working.
The tool will not work if the corresponding Jaeger or Prometheus APIs request or response formats change in a way which is not compatible with the existing Jaeger-backend-datasource plugin's logic.
The tool has not been tested with a significantly larger test project, hence the performance and accuracy cannot be guaranteed for bigger input projects.
Finally, the tool has only been tested with positive test cases. Negative testing has been left out due to shortage of time.

6.2 Limitations

The MsViz tool aims at providing a better visualization of a system of microservices than other opensource tracing and monitoring tools. The tool also includes some of the important metrics from different sources and displays everything in one place. Despite, the ease of aggregation and centralizing the monitoring work, the tool poses some limitations around its output.
Most importantly, the tool cannot automatically extract metrics from the application. All the data comes from already configured Jaeger, Prometheus and MySQL servers. Any inaccuracy in this data will reflect inaccurate output of MsViz microservices call graph.
At this stage, the tool has been made to incorporate the basic and the most important metrics from performance and business perspective, hence, it cannot provide enough information about the overall system of microservices. The managers, developers and operation engineers might still need to navigate around different monitoring and tracing tools as per different needs and tasks.

7 Conclusion

As the IT industry has been moving from monolith to Microservices based architecture, the developers, managers and other stakeholders had to face challenges to make sure the systems meet the expected performance and the users expectations. The major challenge is the observability of such big systems. Despite the availability of opensource monitoring tools, none of them fulfil the need of both the developers and the managers working in financial departments. Different monitoring tools need to be setup and individually monitored as per their outputs and stakeholder's needs.

In this research work, various monitoring tools have been studied to understand the way they work and the kind of output they produce. After performing research on some opensource monitoring tools, three tools Prometheus, MySQL and Jaeger were selected to obtain microservices call dependency and metrics data. Then a tool named MsViz was developed inside Grafana as a combination of different plugins. MsViz uses above three datasources to fetch performance, tracing and business data and metrics and graphically displays them inside a Grafana panel in form of a graph. The metrics are showed in a graphical way inside the microserivces call dependency graph which can help the developers and operation engineers to briefly assess the overall system's health and performance. The Graph can help the managers to briefly predict how the microservices are doing from the business perspective and a deep analysis of the output graph can help the managers make strategic decisions. The tool eases the monitoring process for the stakeholders by aggregating some important data in one place.

Currently, there are various opensource and commercial microservices monitoring and tracing tools capable of providing far deeper and complex information and insights about microservices based systems. Whereas, MsViz tool only incorporates a few important metrics from different opensource tools and combines them to produce a better and visually more powerful graphical presentation. However, the current work shows the future possibility of how this tool can be improved and enhanced to incorporate more metric data and different visualizations can be added for the whole system or individual microservices.

8 References

[1] J. Thöness. Microservices. IEEE Software, 32(1):116–116, 2015.

[2] X. Zhou et al., "Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study," in IEEE Transactions on Software Engineering. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8580420&isnumber=4359463

[3] A. R. Sampaio, H. Kadiyala, B. Hu, J. Steinbacher, T. Erwin, N. Rosa, I. Beschastnikh, J. Rubin, "Supporting Microservice Evolution", 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 539-543, sep 2017.

[4] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," Google, Inc., Tech. Rep., 2010. [Online]. Available: https://research.google.com/archive/papers/dapper-2010-1.pdf

[5] OpenZipkin, "openzipkin/zipkin," 2018. [Online]. Available at: https://github.com/openzipkin/zipkin

[6] Jaeger, "jaegertracing," 2018. [Online]. Available at https://github.com/jaegertracing/jaeger

[7] J. Shaheen, "Apache Kafka: Real Time Implementation with Kafka Architecture Review." International Journal Of Advanced Science And Technology 109 (2017): 35-42.

[8] G. K. Yong. "A data analytic module to extend grafana functionality". Phd Diss. UTAR, 2019.

[9] Dragoni N. et al. (2017) Microservices: Yesterday, Today, and Tomorrow. In: Mazzara M., Meyer B. (eds) Present and Ulterior Software Engineering. Springer, Cham. URL: https://link.springer.com/chapter/10.1007/978-3-319-67425-4_12

[10] K. Bakshi, "Microservices-based software architecture and approaches," 2017

IEEE Aerospace Conference, Big Sky, MT, 2017, pp. 1-8, doi:10.1109/AERO.2017.7943959.

[11] R. BONCEA, A. ZAMFIROIU, I. BACIVAROV, "A scalable architecture for automated monitoring of microservices", Economy Informatics vol. 18, no. 1/2018

[12] M. Cinque, R. D. Corte, A. Pecchia, "Microservices Monitoring with Event Logs and Black Box Execution Tracing", 2018 IEEE Transactions on Services Computing, DOI 10.1109/TSC.2019.2940009

[13] L. Johansson, D. Dossot, "RabbitMQ Essentials - Second Edition". Packt Publishing, 2020, A Rabbit Springs to Life

[14] L. Johansson, D. Dossot, "RabbitMQ Essentials - Second Edition". Packt Publishing, 2020, A Rabbit Springs to LifeQ

[15] D. Rajput. "Hands-On Microservices - Monitoring and Testing". Packt Publishing, 2018, Performance Monitoring of Microservices

[16] D. Rajput. "Hands-On Microservices - Monitoring and Testing". Packt Publishing, 2018, Performance Monitoring of Microservices

[17] 2020. Microservices & Microservice Monitoring. [online] AppDynamics. Available at: https://www.appdynamics.com/solutions/cloud/cloud-monitoring/microservices [Accessed 4 November 2020].

[18] D. Rajput. "Hands-On Microservices - Monitoring and Testing". Packt Publishing, 2018, New Relic

[19] T. Hunter II, "Advanced Microservices: A Hands-on Approach to Microservice Infrastructure and Tooling". Apress, 2017, Monitoring

[20] P. Siriwardena, K. indrasiri. "Microservices for the Enterprise: Designing, Developing, and Deploying". Apress, 2018, Observability

[21] D. Spoonhower, J. Mace, B. Sigelman, R. Isaacs, A. Parker. "Distributed Tracing in Practice". O'Reilly Media, Inc., 2020, Introduction: What Is Distributed Tracing?